Lloyd Fosdick
Guest Editor

Predicative Programming Part I

ERIC C.R. HEHNER

ABSTRACT: Programs are given a new semantics with the merit that a specification written as a first-order predicate can be refined, step by step, to a program via the rules of Predicate Calculus. The semantics allows a free mixture of predicate and programming notations, and manipulation of programs.

0. SPECIFICATIONS

A *specification* of a mechanism should be a full description of the intended observable behavior of the mechanism. For any observation of its actual behavior, it should be clear whether the observation *satisfies* (accords with) the specification. A mechanism is said to *achieve* a specification if every possible observation of its behavior satisfies the specification. It is quite possible and practical to observe a mechanism's failure to achieve a specification (incorrectness); that requires a single, unsatisfactory observation. It is usually impractical or impossible to observe its success (correctness); that usually requires an immense or infinite number of satisfactory observations. The only practical method of determining success is a proof.

In this paper, we consider a mechanism (computer) for which the possible observations are the values of certain variables before it is activated, and if its activity terminates, the values of these same variables after its activity has terminated. This is deliberately a very limited view of computing that excludes the expense and speed of computation, and any input or output during the course of a computation. Execution time will be considered in Section 3, and communication is the subject of Part II. For the present, we consider only an initial input and a final output.

Let x be a variable. We denote the value of x before activation by $\grave{x}$ (pronounced "x in"), and the value of x after termination by $\acute{x}$ (pronounced "x out"). For example, if there are two variables x and y,

$$\acute{x} = \grave{x}+1 \wedge \acute{y}=\grave{y} \tag{0}$$

specifies the behavior of a computer that increases the value of x by 1, and leaves y unchanged. The assertion

$$\grave{x} < \acute{x} \tag{1}$$

also describes this same behavior, but it is a weaker, incomplete description and therefore not a specification of the same behavior.

Assertion (1) can, however, be offered as a specification. As such, it says that the specifier is happy if the computer increases x by any amount, and does not care whether y is changed or left unchanged. Specification (1) is achieved by any mechanism that achieves Specification (0), and also by many mechanisms that do not.

The specification

$$\grave{x} < 0 \vee \acute{x} = 0 \tag{2}$$

is also achieved by a variety of mechanisms. If initially $x \geqslant 0$, then a mechanism's activity must terminate with

This research was supported by the National Science and Engineering Research Council of Canada.

$x = 0$ in order to satisfy the specification. However, if initially $x < 0$, then the specification is already satisfied no matter what the mechanism does: its activity may terminate with x arbitrary, or it may not terminate.

We assume that a person who does not care about the output, i.e., one who does not care about the final value of any variable, also does not care whether the variables have final values, i.e., whether the mechanism's activity terminates. One may object to this assumption, saying that it is possible (although not reasonable) to be interested only in termination but not in any result. But termination is, in effect, a Boolean result; it can be expressed by the introduction of a Boolean variable, say t (for termination). Then

(3) $$(\dot{x} < 0 \vee \acute{x} = 0) \wedge \acute{t}$$

specifies a mechanism with the following behavior: If, initially, $x \geq 0$, then its activity must terminate with $x = 0$ and $t = true$; if, initially, $x < 0$, then its activity must terminate with $t = true$ (and x arbitrary). In either case, the initial value of t is not of interest and is therefore arbitrary. But the final value of t is specified; therefore, termination is required.

Let us use the letter v to mean all the variables of a mechanism. In an assertion, $\dot{v}$ is the vector of initial values, and $\acute{v}$ is the vector of final values of the variables. We can specify that termination is required by indicating interest in the final values of some variables. "Interest" means that not all final values $\acute{v}$ are completely arbitrary. If S is a specification, then

(4) $$\neg\forall\acute{v}.\ S$$

is satisfied by just those initial values for which termination is required. We cannot specify that nontermination is required, nor can we ever observe that nontermination has "occurred". It would not be reasonable to specify unobservable behavior.

Two specifications deserve special mention. One is the weakest of all specifications

(5) $$true$$

It is achieved by every mechanism: those that never terminate, those that terminate for some initial values but not for others, and those that always terminate. In case of termination, the final values of all variables are completely arbitrary. The customer who offers (5) as a specification agrees to accept any mechanism. At the other extreme, the customer who offers

(6) $$false$$

has decided to reject any mechanism. No mechanism achieves (6).

In order to be achievable by some mechanism, a specification must be satisfiable by some observation, as example (6) shows. As the next example shows, this is not enough.

(7) $$\dot{x} = 0$$

Initial values are supplied to the mechanism as input. It must then set the final values so that the specification is satisfied. If the initial value $\dot{x}$ is 0, then (7) is satisfied by any final value $\acute{x}$, but if $\dot{x}$ is not 0, it is impossible to choose $\acute{x}$ to satisfy (7). For a mechanism to achieve (7), all observations of its behavior under all circumstances must satisfy (7). To be achievable, a specification S must obey

(8) $$\forall\dot{v}.\ \exists\acute{v}.\ S$$

This formula introduces an asymmetry into our treatment of initial and final values, thus giving a direction to computation. Even this formula is not enough to ensure that S is achievable.

Specification S_0 is said to be as *determined* as specification S_1 if

(9) $$\forall\dot{v},\acute{v}.\ S_0 \Rightarrow S_1$$

(If this implication is not also equality, then S_0 is more determined than S_1.) Determination is therefore just the partial ordering by implication; (5) is the least determined and (6) the most determined specification. A specification that is so determined that (8) does not hold is said to be *overdetermined.* A specification S is called *deterministic* if

(10) $$\forall\dot{v}.\ \exists 1\acute{v}.\ S$$

which says that for each vector of initial values, there is exactly one vector of final values to satisfy S. A deterministic specification is as determined as possible without being overdetermined, and defines a function from inputs to outputs.

It is not necessary for a specification to be deterministic to be achievable. The necessary and sufficient condition is the following: specification S is achievable if there is a specification DS such that

(a) DS is as determined as S,
(b) DS is deterministic, and
(c) DS defines a computable function.

Although computability is not expressible in a neat formula, it is nonetheless well-defined in standard texts.

1. PROGRAMS

A program in a high-level programming language can be considered as something that controls the behavior of a computer (perhaps indirectly, through a compiler or interpreter). Equally well, a program can be considered as a complete description of the desired observable behavior of a computer. According to the latter view, which we adopt, a program is a specification.

In order to allow specifications to be stated conveniently and in terms appropriate to the intended application, we do not restrict the language of specifications further than this: a specification is an assertion about the initial and final values of some variables. A program is a specification in a highly restricted notation that we present in the next few subsections. The purpose of the restriction is to ensure that a program is an achievable specification, and (thanks to the compiler writers) that a mechanism to achieve it can be built automatically. We do not intend our programming no-

tations to be taken as a complete or exemplary programming language; they were chosen merely to demonstrate our approach to semantics.

1.0 Skip

Our first programming notation is **skip**, defined as

$$\textbf{skip} =_{df} \dot{v}=\grave{v}$$

For example, if there are two variables x and y, then

$$\textbf{skip} = (\acute{x}=\grave{x} \wedge \acute{y}=\grave{y})$$

The notation **skip** specifies that all variables have the same final values as initial values. The fastest and cheapest mechanism that achieves **skip** is the mechanism that does nothing whatever.

1.1 Assignment

Our next programming notation is the familiar assignment $x := e$ where x is a variable and e is an expression. To define it, some auxiliary notions are needed.

Let e and f be any two expressions, and let x be any variable. Then f^x_e is the expression formed from f by replacing every free occurrence of x by e. (If this substitution would place free variables of e in a bound context, the offending bound variables of f must first be renamed.)

Let e be any expression in the program variables v. Then $\grave{e}$ is e but with ` over each program variable. Similarly, $\acute{e}$ is obtained from e by placing ´ over each program variable.

$$\grave{e} =_{df} e^{\grave{v}}_{v}$$

$$\acute{e} =_{df} e^{\acute{v}}_{v}$$

Let e be any expression in some programming language. Then D‘$\grave{e}$’ asserts that expression $\grave{e}$ is defined, i.e., that all variables have initial values, and that all operands are within the domain of their operators. It is defined according to the structure of $\grave{e}$. We shall not be more precise about the syntax or semantics of expressions, but here are three examples:

D‘1’	is true, and similarly for all constants.
D‘$\grave{x}$’	is irreducible, asserting that variable x is initialized.

$$D‘\grave{x}/\grave{y}’ = D‘\grave{x}’ \wedge D‘\grave{y}’ \wedge \grave{y}\neq 0$$

Assignment can now be defined as

$$x{:=}e =_{df} (D‘\grave{e}’ \wedge \grave{e}\in \text{type}(x)) \Rightarrow (\dot{v}=\grave{v}^{\grave{x}}_{\grave{e}})$$

The type of a variable is obtained from its declaration; we henceforth ignore questions of type, always assuming type-correctness. Here are two examples, each using the two variables x and y.

$$x := x/y$$

$$= (D‘\grave{x}’ \wedge D‘\grave{y}’ \wedge \grave{y}\neq 0) \Rightarrow (\acute{x}=\grave{x}/\grave{y} \wedge \acute{y}=\grave{y})$$

$$x := x{-}x$$

$$= D‘\grave{x}{-}\grave{x}’ \Rightarrow (\acute{x} = \grave{x}{-}\grave{x} \wedge \acute{y}=\grave{y})$$

$$= D‘\grave{x}’ \Rightarrow (\acute{x}=0 \wedge \acute{y}=\grave{y})$$

In the last example, x must be initialized in order to conclude anything about the final values, even though the final values do not depend on the initial value of x.

When D‘$\grave{e}$’ is false, the specification $x := e$ is satisfied, and every mechanism achieves it. The preferred mechanism is one that gives a helpful error indication, but unfortunately we cannot insist on that mechanism. If we were to define assignment so that an error indication is required when $\grave{e}$ is undefined, the "halting problem" tells us that some programs would be unachievable.

1.2 Composition

Let P and Q be specifications. Then P;Q is a specification that can be achieved by a mechanism behaving as follows: it first satisfies P; if it terminates, it then satisfies Q with the final values from P serving as initial values for Q. The phrase "if it terminates" becomes, according to formula (4), "$(\neg\forall\dot{v}.\ P) \Rightarrow \ldots$". Let $\dot{v}$ be fresh names (i.e., $\dot{x}$, $\dot{y}$, ...) denoting the intermediate values of the variables between P and Q. Then

$$P;Q =_{df} \exists\dot{v}.\ P^{\dot{v}}_{v} \wedge ((\neg\forall\dot{v}.\ P) \Rightarrow Q^{\dot{v}}_{v})$$

or equivalently

$$P;Q =_{df} (\neg\forall\dot{v}.\ P) \Rightarrow (\exists\dot{v}.\grave{P}^{\dot{v}}_{v} \wedge Q^{\grave{v}}_{v})$$

We have introduced ";" as a new logical connective, like "$\wedge$" and "$\vee$" that joins any two specifications. If the two specifications P and Q happen to be programs (they use only programming notations), then "P;Q" is also a program.

Composition is most easily used via the following theorem.

THEOREM 0.

(a) **skip**;P = P;**skip** = P

(b) $(x := e;\ P) = (D‘\grave{e}’ \Rightarrow P^{\grave{x}}_{\grave{e}})$

PROOF. predicate calculus.

In part (b), it should be stressed that substitutions are not made in the programming notation, but in the assertions which we have given as the meaning of programs.

The next theorem gives further properties of composition that will be useful later, and points out a technical difficulty: semicolon is not associative for all predicates.

THEOREM 1.

(a) *true* ; P

(b) $(\exists\dot{v}.\ P) \Rightarrow (P;\mathit{true})$

(c) $P;(Q;R) \Rightarrow (P;Q);R$

(d) $((\forall\dot{v}.\ P) \vee (\exists 1\dot{v}.\ P)) \Rightarrow (P;(Q;R) = (P;Q);R)$

PROOF. predicate calculus.

The hypothesis of Theorem 1(b) says that, for a given input, P is not overdetermined; all our programming notations satisfy that hypothesis. The hypothesis of part (d) says that for a given input, P is either undetermined or deterministic; all but one (nondeterministic choice,

$$
\begin{aligned}
& y := y-1;\ x := x/y \\
&= y := y-1;\ ((D`\dot{x}` \wedge D`\dot{y}` \wedge \dot{y}\neq 0) \Rightarrow (\acute{x}=\dot{x}/\dot{y} \wedge \acute{y}=\dot{y})) \\
&= D`\dot{y}-1` \Rightarrow ((D`\dot{x}` \wedge D`\dot{y}` \wedge \dot{y}\neq 0) \Rightarrow (\acute{x}=\dot{x}/\dot{y} \wedge \acute{y}=\dot{y}))^{\dot{y}}_{\dot{y}-1} \\
&= D`\dot{y}` \Rightarrow ((D`\dot{x}` \wedge D`\dot{y}-1` \wedge \dot{y}-1\neq 0) \Rightarrow (\acute{x}=\dot{x}/(\dot{y}-1) \wedge \acute{y}=\dot{y}-1)) \\
&= (D`\dot{x}` \wedge D`\dot{y}` \wedge \dot{y}\neq 1) \Rightarrow (\acute{x}=\dot{x}/(\dot{y}-1) \wedge \acute{y}=\dot{y}-1) \\
& y := 2;\ y := y-1;\ x := x/y \\
&= D`2` \Rightarrow ((D`\dot{x}` \wedge D`\dot{y}` \wedge \dot{y}\neq 1) \Rightarrow (\acute{x}=\dot{x}/(\dot{y}-1) \wedge \acute{y}=\dot{y}-1))^{\dot{y}}_{2} \\
&= \mathit{true} \Rightarrow ((D`\dot{x}` \wedge D`2` \wedge 2\neq 1) \Rightarrow (\acute{x}=\dot{x}/(2-1) \wedge \acute{y}=2-1)) \\
&= D`\dot{x}` \Rightarrow (\acute{x}=\dot{x} \wedge \acute{y}=1)
\end{aligned}
$$

FIGURE 1.

Section 1.4) of our programming notations satisfy that hypothesis. On the rare occasions when it matters, we consider that ";" associates to the right, i.e., (P;Q;R) means (P;(Q;R)).

The difficulty with semicolon is theoretically annoying, but not really an impediment, as the next theorem shows.

THEOREM 2.

(a) Let P, Q, and R be programs over an infinite state space. Then P;(Q;R) = (P;Q);R.

(b) Let P, Q, and R be programs, and let there be at least one variable that does not appear in P or Q. Then P;(Q;R) = (P;Q);R.

PROOF. structural induction.

According to Theorem 2(a), one integer variable is enough to ensure associativity. According to Theorem 2(b), one unused variable is enough.

Figure 1 shows some examples in two variables.

1.3 Deterministic Choice

Let P and Q be specifications, and let b be a Boolean expression in the program variables. Then the programming notation

if b **then** P **else** Q

specifies behavior that satisfies either P or Q depending on whether b is initially true or false.

if b **then** P **else** Q $=_{df}$ $D`\dot{b}` \Rightarrow (\dot{b} \wedge P \vee \neg\dot{b} \wedge Q)$

Here is an example in two variables.

if $x \leq y$ **then** $x := y$ **else skip**

$= D`\dot{x}\leq\dot{y}` \Rightarrow (\dot{x}\leq\dot{y} \wedge x := y \vee \dot{x}>\dot{y} \wedge$ **skip**$)$

$= (D`\dot{x}` \wedge D`\dot{y}`) \Rightarrow (\dot{x}\leq\dot{y} \wedge \acute{x}=\acute{y}=\dot{y} \vee \dot{x}>\dot{y} \wedge \acute{x}=\dot{x} \wedge \acute{y}=\dot{y})$

$= (D`\dot{x}` \wedge D`\dot{y}`) \Rightarrow (\acute{x}=\max(\dot{x}, \dot{y}) \wedge \acute{y}=\dot{y})$

THEOREM 3.

(a) $D`\dot{b}` \Rightarrow$ ((**if** b **then** P **else** P) = P)

(b) (**if** b **then** P **else** Q) = (**if** $\neg b$ **then** Q **else** P)

(c) ((**if** b **then** P **else** Q); R)
= (**if** b **then** (P; R) **else** (Q;R))

PROOF. predicate calculus.

1.4 Nondeterministic Choice

Let P and Q be specifications. Then the programming notation

P **or** Q

specifies behavior that satisfies either P or Q.

P **or** Q $=_{df}$ P $\vee$ Q

No criterion for making the choice is offered, so an implementer has complete freedom to choose either. This freedom can be used to choose the faster, or cheaper, of the two. The choice may be made before execution, or during execution, according to the values of some variables or whether one mechanism is broken and the other working.

THEOREM 4.

(a) (P **or** Q) **or** R = P **or** (Q **or** R)

(b) P **or** Q = Q **or** P

(c) P **or** P = P

(d) *true* **or** P

(e) P;(Q **or** R) = (P;Q) **or** (P;R)

(f) $((\forall \acute{v}.\ P) \vee (\forall \acute{v}.\ Q) \vee (\neg\forall \acute{v}.\ P \vee Q))$
$\Rightarrow$ ((P **or** Q);R = (P;R) **or** (Q;R))

(g) (**if** b **then** (P **or** Q) **else** (P **or** R))
= P **or** (**if** b **then** Q **else** R)

PROOF. predicate calculus.

Without nondeterministic choice, our other programming notations allow us to express deterministic specifications and completely undetermined specifications (e.g., $x := 1/0$), but nothing between. With **or**, our programming notations allow us to express bounded nondeterminism, the property of Theorem 6.

THEOREM 5.

If P is a program without **or**, then

(a) $(\forall \acute{v}.\ P) \vee (\exists 1 \acute{v}.\ P)$

(b) P;(Q;R) = (P;Q);R

PROOF. for (a), induction on the structure of P; for (b), Theorems 1(d) and 5(a).

THEOREM 6.
If P is a program, then

$$(\forall \acute{v}.\ P) \vee (\exists \acute{v}.\ P)$$

where "$\exists \acute{v}$. P" means "there is at least one, and at most a finite number of $\acute{v}$ satisfying P".

PROOF. structural induction.

COROLLARY. formula (8).

1.5 Definition

If a program is larger than a few lines, it is helpful to give descriptive names to parts of the program, and then use the names in place of the parts. A well-chosen name reminds us of the purpose, or meaning, of the part being named. And when a lengthy part occurs repeatedly, a short (but descriptive) name is also an abbreviation. The ability to name a program and to use a name in place of a program (in a larger program) provides a structure for large programs that aids our understanding of them and our ability to compose them. Our naming, or definition, notation is

name: program

A definition is not a program, but sits near a program to give meaning to the name, which may then be used in the program. (A programming language must say where definitions are allowable and what their scope is, but in this paper we do not care to be more explicit.)

The following example does not illustrate the importance of definitions in large programs, but it illustrates two points in their use. The definitions

$$negx\text{: } x := -x$$

$$absx\text{: } \textbf{if } x<0 \textbf{ then } negx \textbf{ else skip}$$

allow us to write

$$\begin{aligned}
& x := y;\ absx \\
&= D\dot{y}' \Rightarrow absx^{\dot{x}}_{\dot{y}} \\
&= D\dot{y}' \Rightarrow (\textbf{if } \dot{x}<0 \textbf{ then } negx \textbf{ else skip})^{\dot{x}}_{\dot{y}} \\
&= D\dot{y}' \Rightarrow (D\dot{x}<0' \Rightarrow ((\dot{x}<0 \wedge negx) \vee (\dot{x}\geq 0 \wedge \textbf{skip})))^{\dot{x}}_{\dot{y}} \\
&= D\dot{y}' \Rightarrow (D\dot{x}' \Rightarrow (\dot{x}<0 \wedge x:=-x) \\
&\qquad\qquad \vee (\dot{x}\geq 0 \wedge \acute{x}=\dot{x} \wedge \acute{y}=\dot{y}))^{\dot{x}}_{\dot{y}} \\
&= D\dot{y}' \Rightarrow (D\dot{x}' \Rightarrow (\dot{x}<0 \wedge (D\dot{x}' \Rightarrow \acute{x}=-\dot{x} \wedge \acute{y}=\dot{y})) \\
&\qquad\qquad \vee (\dot{x}\geq 0 \wedge \acute{x}=\dot{x} \wedge \acute{y}=\dot{y}))^{\dot{x}}_{\dot{y}} \\
&= D\dot{y}' \Rightarrow (D\dot{y}' \Rightarrow (\dot{y}<0 \wedge \acute{x}=-\dot{y} \wedge \acute{y}=\dot{y}) \\
&\qquad\qquad \vee (\dot{y}\geq 0 \wedge \acute{x}=\dot{y} \wedge \acute{y}=\dot{y})) \\
&= D\dot{y}' \Rightarrow (\acute{x}=-\dot{y}>0 \vee \acute{x}=\dot{y}\geq 0) \wedge \acute{y}=\dot{y}
\end{aligned}$$

First, notice that program names may be used in definitions. Second, notice that the order in which names are replaced by programs, and programs by traditional predicate notations, does not matter. However, variable substitutions are made after these replacements in the traditional predicate notations.

If a name is used in its own definition, either directly or indirectly via other definitions, the definition is said to be recursive. No finite sequence of replacements can rid a program of a recursively defined name; nonetheless, it is the infinite sequence of replacements that gives meaning to a recursively defined name. Let F(P) be a program in which P occurs zero or more times. Then the definition

P: F(P)

gives P the meaning

$$P =_{df} \forall i.\ F^i(true)$$

where

$$F^0(p) =_{df} p$$

$$F^{i+1}(p) =_{df} F(F^i(p))$$

The justification for this formula is based on the following theorem.

THEOREM 7.
If F(P) is a program and $P_0, P_1, P_2, \ldots$ is a monotonically strengthening sequence of programs, i.e., $\forall i.\ P_{i+1} \Rightarrow P_i$, then

(a) $\forall i.\ F(P_{i+1}) \Rightarrow F(P_i)$
(b) $(\forall i.\ F(P_i)) = F(\forall i.\ P_i)$

PROOF. Induction on the structure of F.

This theorem says that programs are (a) monotonic and (b) continuous functions of any names occurring in them.

The recursive definition P: F(P) defines P in terms of itself; thus it may seem that we need to know what P means before we can determine what P means. Knowing only that P is a program, and nothing about the mechanism it specifies, we can say only this: the mechanism is described by the assertion *true* (as all mechanisms are). Let P_0 be this initial, least-determined description.

$$P_0 = true$$

Now that we have a description, we can obtain another by using P_0 in place of P in F(P). Our second description of the mechanism is

$$P_1 = F(P_0)$$

Continuing, we form a sequence of descriptions

$$P_{i+1} = F(P_i)$$

The following theorem tells us that this sequence is monotonically strengthening, and therefore the successive descriptions are more and more determined.

THEOREM 8.
If F(P) is a program, then $F^i(true)$ is a monotonically strengthening sequence, i.e., $\forall i.\ F^{i+1}(true) \Rightarrow F^i(true)$.

PROOF. induction on i, and Theorem 7(a).

We take the limit of the sequence P_i to be an exact description, or specification, of the mechanism. The limit of a strengthening sequence is the conjunction of its members. Each P_i is a description; we consider that all P_i together constitute a complete description.

$$P =_{df} \forall i.\ P_i$$

Our confidence that this is the appropriate meaning for P comes from the following two theorems.

THEOREM 9.
If P: F(P) is a program definition, then P = F(P).

PROOF: Knaster and Tarski, and Theorem 7(b).

So a name can be replaced by the program it stands for without changing the meaning, for recursively defined names as well as nonrecursively defined names.

THEOREM 10.
If P: F(P) is a program definition and Q = F(Q), then Q⇒P.

PROOF: induction on P_i.

Our meaning for P makes it as general as possible: P is achieved by all and only those mechanisms with the replacement property (Theorem 9).
Here is an example.

xyz: **if** $x=0$ **then** $y:=0$

else $(x:=x-1;\ xyz)$

The successive descriptions are

$$xyz_0 = true$$

$$xyz_{i+1} = (\mathrm{D}`\dot{x}` \Rightarrow \dot{x}=\acute{x}=\acute{y}=0 \vee (\dot{x}\neq 0 \wedge xyz_i\ {}^{\dot{x}}_{\dot{x}-1}))$$

By finding the first few predicates of this sequence we are led to propose

$$xyz_i = (\mathrm{D}`\dot{x}` \wedge 0\leq\dot{x}<i) \Rightarrow \acute{x}=\acute{y}=0$$

and to prove it by induction. Thus we know

$$xyz = \forall i.\ xyz_i$$

$$= (\mathrm{D}`\dot{x}` \wedge 0\leq\dot{x}) \Rightarrow \acute{x}=\acute{y}=0$$

If initially $\dot{x}<0$, then final values are not of interest and termination is not required; if initially $0\leq\dot{x}$, then termination is required with final values $\acute{x}=\acute{y}=0$.

Indirect recursion is an easy generalization of direct recursion. For example, the definitions

$$P: F(P, Q)$$

$$Q: G(P, Q)$$

mean

$$P =_{df} \forall i.\ P_i$$

$$Q =_{df} \forall i.\ Q_i$$

where

$$P_0 = Q_0 = true$$

$$P_{i+1} = F(P_i, Q_i)$$

$$Q_{i+1} = G(P_i, Q_i)$$

Iterative constructs are easily defined as special cases of recursive definition. For example

W: **while** *b* **do** P

means

W: **if** *b* **then** P;W **else skip**

and

R: **repeat** P **until** *b*

means

R: P; **if** *b* **then skip else** R

1.6 Declaration

Declaration is a means of introducing new variables to local parts of a program, so that not all parts need be concerned with all variables. Our notation for introducing variable x locally within P is

var x. P

(We continue to avoid the issue of types. Type information provided in a declaration is used in assignments to the declared variable, not in the assertion that declares a variable to be local.)

The variable introduced by a declaration is a new one, distinct from all existing variables. It is usual, and useful in programming, to allow the newly introduced (local) variable to have the same name as an existing (global) variable, in which case the local variable temporarily obscures the global one. To express the program **var** x. P as an assertion about the initial and final values $\dot{v}$ and $\acute{v}$ of all the global variables v, the local variable must be renamed if necessary to avoid obscuring a global variable.

Our declaration introduces a variable with no initial value. We express this fact by a standard technical trick: let $\perp$ (called "the undefined value") have the one and only property

$$\neg \mathrm{D}`\perp`$$

Declaration can now be defined as

$$\textbf{var } x.\ P =_{df} \exists \acute{x}.\ P^{\dot{x}}_{\perp}$$

In the following examples, the global variables are x and y.

$$\textbf{var } z.\ z := 1;\ y := z$$
$$= \exists \acute{z}.\ (\mathrm{D}`1` \Rightarrow (\mathrm{D}`\dot{z}` \Rightarrow (\acute{x}=\dot{x} \wedge \acute{y}=\dot{z} \wedge \acute{z}=\dot{z}))^{\dot{1}}_{\dot{1}})^{\dot{z}}_{\perp}$$
$$= \exists \acute{z}.\ (true \Rightarrow (\mathrm{D}`1` \Rightarrow (\acute{x}=\dot{x} \wedge \acute{y}=1 \wedge \acute{z}=1)))^{\dot{1}}_{\dot{1}}$$
$$= \exists \acute{z}.\ \acute{x}=\dot{x} \wedge \acute{y}=1 \wedge \acute{z}=1$$
$$= \acute{x}=\dot{x} \wedge \acute{y}=1$$
$$\textbf{var } z.\ x := z-z$$
$$= \exists \acute{z}.\ (\mathrm{D}`\dot{z}-\dot{z}` \Rightarrow \acute{x}=\dot{z}-\dot{z} \wedge \acute{y}=\dot{y} \wedge \acute{z}=\dot{z})^{\dot{z}-\dot{z}}_{\perp}$$
$$= \exists \acute{z}.\ (\mathrm{D}`\dot{z}` \Rightarrow \acute{x}=0 \wedge \acute{y}=\dot{y} \wedge \acute{z}=\dot{z})^{\dot{z}}_{\perp}$$
$$= \exists \acute{z}.\ \mathrm{D}`\perp` \Rightarrow \acute{x}=0 \wedge \acute{y}=\dot{y} \wedge \acute{z}=\perp$$
$$= \exists \acute{z}.\ false \Rightarrow \acute{x}=0 \wedge \acute{y}=\dot{y} \wedge \acute{z}=\perp$$
$$= \exists \acute{z}.\ true$$
$$= true$$

A program can be rewritten as an assertion in the initial and final values of individual variables only when its context is given. For example, in

skip; **var** z. z:=1; **skip**; y:=z

the first occurrence of skip means $\dot{x}=\grave{x} \wedge \dot{y}=\grave{y}$ but the second occurrence means $\dot{x}=\grave{x} \wedge \dot{y}=\grave{y} \wedge \dot{z}=\grave{z}$. The same is true for defined names. The definition

P: **skip**

gives P the meaning $\dot{v}=\grave{v}$, but to be more specific we must look at the contexts in which P is used. In

P; **var** z. z:=1; P; y:=z

the first occurrence of P means $\dot{x}=\grave{x} \wedge \dot{y}=\grave{y}$ and the second occurrence means $\dot{x}=\grave{x} \wedge \dot{y}=\grave{y} \wedge \dot{z}=\grave{z}$.

According to the following theorem, the order of declarations does not matter.

THEOREM 11.

var x. (**var** y. P) = **var** y. (**var** x. P)

PROOF: predicate calculus.

2. PROOFS AND PROGRAMMING

Program P is said to be correct for specification S if

$$\forall \grave{v},\dot{v}.\ D'\grave{v}' \wedge P \Rightarrow S$$

This means that when input is supplied, P is as determined as S, and every mechanism achieving P also achieves S. Section 1 provides us with a calculus for proving whether a given program is correct for a given specification.

The programmer's task is different; only a specification is given, and the task is to construct a program that is correct, if one exists. The calculus required is a kind of inverse of the calculus presented in Section 1.

Since a program is a specification, its notations can be mixed with nonprogram notations via the usual connectives of predicate calculus. For example, if P is a program and S is a specification (which may or may not be a program), then P $\wedge$ S is a specification of computer behavior that satisfies both P and S. It may not be achievable, and it is not a program (since "$\wedge$" is not one of our programming notations), but it is a specification. The ability to mix program and nonprogram notations in one specification allows us to begin with a specification in a form that is convenient to the specifier, and convert it little by little to a program, using the rules of predicate calculus. After each step in this conversion, the specification should be as determined (strong) as it was before (but not overdetermined), in order that the final program will be correct for the original specification. Judicious strengthening can be used to resolve nondeterminacy and narrow the choice of mechanisms.

A specification is sometimes given by a precondition G on the initial values and postcondition R on the final values ("G" for "Given", "R" for "Result"). As a single predicate, the specification is

(11) $\qquad \dot{G} \Rightarrow \dot{R}$

One way to refine this specification is to use the next theorem.

THEOREM 12.
Let G, I, and R be any predicates in the program variables.

$$((\neg\ \forall v.\ I) \wedge (\exists v.\ I)) \Rightarrow ((\dot{G} \Rightarrow \dot{R}) = (\dot{G} \Rightarrow \acute{I})\ ;\ (\grave{I} \Rightarrow \dot{R}))$$

PROOF: predicate calculus.

We can choose any predicate I (for "Intermediate" or "Invariant") that is neither identically true (undetermined) nor identically false (overdetermined), and replace (11) by the equivalent, but more refined,

$$(\dot{G} \Rightarrow \acute{I})\ ;\ (\grave{I} \Rightarrow \dot{R})$$

giving us two new problems of the same form. Choosing a good I can be difficult: the essence of creative programming.

Definitions can be made freely, provided that for each loop (recursion), there is an integer expression e (the variant) such that

$$\forall \grave{v},\acute{v}.\ L \wedge 0<\grave{e} \Rightarrow 0 \le \acute{e} < \grave{e}$$

where L consists of the statements in the path of the loop.

3. SEQUENTIAL EXECUTION TIME

Suppose that we know the sequential execution time required for each expression evaluation and assignment in a program P. (This information can be supplied by an implementation, or an assumption can be made.) From this information, using the calculus we have presented, sequential execution time bounds can be proven. Choose a fresh variable, one not occurring in P, say t, to stand for the sequential execution time. Create a new program Pt as follows. For each assignment, include another assignment $t := t + a$ where a is its execution time, either just before or just after it. For each deterministic choice, include the assignment $t := t+c$ where c is the time to evaluate the Boolean expression, either before or after the choice statement. To prove that $f(\grave{v})$ is an upper bound for the sequential execution time, we must prove

(12) $\qquad (Pt \wedge \grave{t}=0) \Rightarrow (\acute{t} \le f(\grave{v}))$

If initially the time is 0, and it is increased as described by Pt, then finally it will be at most $f(\grave{v})$. To prove that $f(\grave{v})$ is a lower bound, we must prove

(13) $\qquad (Pt \wedge \grave{t}=0) \Rightarrow (\acute{t} \ge f(\grave{v}))$

Here is a simple example. The program is P, defined as

P: **if** n=0 **then skip else** (n := n−1; P)

Let the time to evaluate (n = 0) be c, and the time for the assignment (n := n−1) be a. Then

Pt: t := t+c; **if** n=0 **then skip**
else (t := t+a; n := n−1; Pt)

We want to prove that

$$f(\grave{n}) = (c+a)\grave{n}+c$$

is an upper bound for execution time. Thus we want

(14) $\quad (\mathrm{Pt} \wedge \grave{t}=0) \Rightarrow (\grave{t} \leq (c+a)\grave{n}+c)$

For this simple example, we can find

(15) $\quad \mathrm{Pt} = (\grave{n} \geq 0 \Rightarrow (\acute{n}=0 \wedge \acute{t} = \grave{t}+(c+a)\grave{n}+c))$

from which (14) can be simplified to

$$\grave{n} \geq 0$$

This says that $f(\grave{n})$ is an upper bound for the execution time of P provided $\grave{n} \geq 0$. (Nothing whatever is said of the execution time when $\grave{n}<0$.)

In general, it is very difficult, perhaps impossible, to express Pt as a simple, traditional predicate. Fortunately, it is not necessary to do so. What must be proven is an implication (12); its antecedent can be weakened without harm. In particular, conjuncts of Pt that do not refer to time (in (15) conjunct $\grave{n}=0$) can be dropped. No contribution to complexity theory is being made here; we only wish to show that the problem can be stated, and in principle solved, in the calculus we have presented.

4. SEMANTIC DISTINCTIONS

It is not our purpose to describe arbitrary mechanisms, but to prescribe desired ones. Accordingly, our semantic formalism is just powerful enough to distinguish the desirable mechanisms:

(a) must terminate with a desired result,

from the undesirable ones:

(b) might not terminate with a desired result.

(In Part II, we introduce communication so that results can be obtained during a computation. There, a nonterminating computation may be desired. But here in Part I, results are obtained only upon termination.)

A more discriminating formalism, e.g., Dijkstra's wp, is required if we wish to distinguish the following two subclasses of class (b):

(b.a) must terminate, but result might not be a desired one,
(b.b) might not terminate.

A still more discriminating formalism, e.g., one that introduces the poststate $\perp$, is required if we wish to distinguish

(b.b.a) might not terminate, but also might terminate,
(b.b.b) must not terminate.

From the prescriptive viewpoint, it is no demerit of Dijkstra's wp that it cannot distinguish (b.b.a) from (b.b.b). Similarly, we consider it no demerit of our semantics that it cannot distinguish (b.a) from (b.b).

We introduced the notation D'$\grave{x}$' to mean that variable x has an initial value. We could have introduced the similar notation D'$\acute{x}$' to mean that x has a final value. For example, assignment could be defined as

$$x{:=}e = (\mathrm{D}'\grave{e}' \Rightarrow (\mathrm{D}'\acute{x}' \wedge \acute{v}{=}\grave{v}^{\grave{x}}_{e}))$$

In Section 2, D'$\grave{v}$' was used to mean "all variables have initial values". The appropriate meaning for D'$\acute{v}$' would be "some variable has a final value."

$$\mathrm{D}'\grave{v}' = \mathrm{D}'\grave{x}' \wedge \mathrm{D}'\grave{y}' \wedge \cdots$$

$$\mathrm{D}'\acute{v}' = \mathrm{D}'\acute{x}' \vee \mathrm{D}'\acute{y}' \vee \cdots$$

We could then distinguish between completely undetermined behavior (*true*) and terminating behavior (D'$\acute{v}$'), without saying anything more specific about final values. But we chose a less elaborate, less powerful formalism.

5. PREVIOUS AND RELATED WORK

The present work is a direct descendant of the seminal papers by Floyd [5] and Hoare [6] which introduced the idea of describing programs and algorithms with predicate assertions, and proving things about programs.

The understanding of recursive definitions comes from Scott [11]. In that work, programs are ordered by "definedness", a concept introduced for the purpose; our ordering is standard predicate implication. For the source of Theorem 9, see [9].

A semantics for a similar programming language fragment is presented in work by Dijkstra [2, 3]. According to the semantics presented there, a program is, in essence, a function from predicates to predicates. Our semantics is of lower order: a program is a predicate. The relationship between them is

$$\mathrm{P} = \neg \mathrm{wp}(\mathrm{P}, v \neq \grave{v})$$

except in the rare circumstances when our semicolon is not associative.

It has been the goal of at least three distinct research efforts to be able to reason directly, mathematically, in the programming notation: the functional programming of Backus [1], the LUCID project of Ashcroft and Wadge [0], and the PROLOG project of van Emden and Kowalski [4]. In spirit, the PROLOG project is closest to the present work, choosing the language of predicates for reasoning. They have designed their programming notations to be a subset of the standard notation for predicates. We have interpreted standard programming notations as predicates.

6. CONCLUSION

Descriptions of a mechanism are assertions about the mechanism; so that we can reason effectively about the mechanism, an appropriate language in which to express our assertions is the language of predicate logic. A specification is a description that is complete in the sense that it describes everything of interest about the mechanism. A program is a specification of computer behavior. Thus we are led to consider programs as assertions in predicate logic.

The biggest disappointment of the approach is that composition (semicolon) is not associative for all predicates (see the appendix). The problem does not exist for semantics of higher order, in which a program is a function from predicates to predicates, and semicolon is

functional composition. But higher-order semantics are considerably more complicated: intuitively, a program cannot so directly be considered as a description of a mechanism; formally, a program must be monotonic and continuous not only in predicate variables, but also in predicate function variables.

The real test of this (or any) semantics is the help it provides a programmer who cares to be rigorous. The higher-order semantics has already proved itself in this respect. Our semantics looks very promising because it allows a free mixture of programming and logical connectives; programs and specifications are on the same level. But it has yet to be proved by experience.

Our semantic formalism is particularly helpful when comparing alternative semantics. For example, we could have made the definitions

$$x := e =_{df} (D[\grave{e}] \Rightarrow \acute{x}=\grave{e}) \wedge \acute{y}=\grave{y} \wedge \cdots$$

$$P;Q =_{df} \exists \grave{v}.\ P^{\acute{v}}_{\grave{v}} \wedge Q^{\grave{v}}_{\acute{v}}$$

These definitions require an implementation technique known as "lazy evaluation" or "output-driven data flow". Since these definitions are stronger than the ones we have chosen, any implementation of these definitions is also an implementation of our chosen definitions. Some new theorems can be proven, while some old ones, such as Theorem 6, become false.

Another advantage of our semantics, and a motivation for its development, is that it allows us to integrate communicating processes with variables and assignments. That is the subject of Part II.

APPENDIX

By assuming the existence of a variable that does not appear in any program, Theorem 2(b) tells us that composition is associative for all programs. By adding a variable to our semantic formalism, we can obtain a predicate semantics in which composition is associative for all predicates, not just those that are, or are equivalent to, programs. We now present that alternative.

As an aid to physical intuition, consider that a computer is operated as follows. We first assign the variables $x, y, \ldots$ their initial values $\grave{x}, \grave{y}, \ldots$. Then we push the start button. We wait until the stop light is lit (this may be an infinite wait). After the stop light is lit, we can observe the final values $\acute{x}, \acute{y}, \ldots$.

Let s be a Boolean variable, but not one of the variables $x, y, \ldots$ that can appear in programs. Let v be the vector of variables $s, x, y, \ldots$. Let $\grave{v}$ be the vector of initial values $\grave{s}, \grave{x}, \grave{y}, \ldots$, and let $\acute{v}$ be the vector of final values $\acute{s}, \acute{x}, \acute{y}, \ldots$ ($\grave{s}$ corresponds to the proposition "the start button is pushed", and $\acute{s}$ corresponds to the proposition "the stop light is lit"). We now redefine three of our programming notations.

$$\textbf{skip} =_{df} \grave{s} \Rightarrow \acute{v}=\grave{v}$$

$$x:=e =_{df} D[\grave{e}] \wedge \grave{s} \Rightarrow \acute{v}=\grave{v}^{\grave{x}}_{\grave{e}}$$

$$P;Q =_{df} \exists \grave{v}.\ P^{\acute{v}}_{\grave{v}} \wedge Q^{\grave{v}}_{\acute{v}}$$

For example, in variables x and y,

$$\textbf{skip} = (\grave{s} \Rightarrow \acute{s}=\grave{s} \wedge \acute{x}=\grave{x} \wedge \acute{y}=\grave{y})$$

We leave the definitions of our other programming notations unchanged.

Every achievable specification can be put in the form

$$D \wedge \grave{s} \Rightarrow \acute{s} \wedge R$$

where D is a predicate in only the initial values $\grave{x}, \grave{y}, \ldots$ and R is a predicate relating initial values $\grave{x}, \grave{y}, \ldots$ to final values $\acute{x}, \acute{y}, \ldots$ such that

$$\forall \grave{x}, \grave{y}, \ldots\ \exists \acute{x}, \acute{y}, \ldots\ D \Rightarrow R$$

D is the "domain of proper termination", and R is the desired input–output relation on that domain. This pair of predicates is used for specifications by Jones [8] and by Parnas [10].

Theorems 1(c, d) and 4(f) can now be strengthened to

$$P;(Q;R) = (P;Q);R$$

$$(P \textbf{ or } Q);R = (P;R) \textbf{ or } (Q;R)$$

Against these gains, there are some small losses. Theorems 0(a), 0(b) and 1(a) need to be weakened by stipulating that P must be a program; or for a general predicate P they must be weakened to

$$\grave{s} \Rightarrow (\textbf{skip};P = P;\textbf{skip} = P)$$

$$x:=e;P = (D[\grave{e}] \wedge \grave{s} \Rightarrow P^{\grave{x}}_{\grave{e}})$$

$$(\exists \grave{v}.\ P) \Rightarrow (\mathit{true};P)$$

And there is a larger, practical loss: all specifications are burdened with occurrences of $\grave{s}$ and $\acute{s}$.

Acknowledgments C.A.R. Hoare provided insights and criticisms, suggestions and guidance, and wrote some early drafts from which the opening paragraph was taken. He also suggested a way of making semicolon associative, which is described in the appendix. I also thank IFIP Working Group 2.3 (Programming Methodology), Mark Saaltink, and Dan Craigen for their comments.

REFERENCES

0. Ashcroft, E.A. and Wadge, W.W. LUCID—A formal system for writing and proving programs. *Siam J. Comp. 5* (1976), 336–354.
1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
2. Dijkstra, E.W. Guarded commands, nondeterminacy, and a formal derivation of programs. *Commun. ACM 18* (1975), 453–458.
3. Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.
4. van Emden, M.H., Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (1976), 733–742.
5. Floyd, R.W. Assigning meaning to programs. *Math. Aspects of Comput. Sci. 19*, (1967), 19–32.
6. Hoare, C.A.R. An axiomatic approach to computer programming. *Commun. ACM 12* (1969), 576–580, 583.
7. Hoare, C.A.R. Is there a mathematical basis for computer programming? *NAG Newsletter 2* (Sept. 1981), 6–14.
8. Jones, C.B. *Software Development: A Rigorous Approach.* Prentice-Hall International, London, 1980.

9. Lassez, J.-L., Nguyen, V.L., Sonenburg, E.A. Fixed-point theorems and semantics: A folk tale. *Inf. Proc. Let. 14*, 3 (May 16, 1982), 112–116.
10. Parnas, D.L. A generalized control structure and its formal definition. *Commun. ACM 26*, 8 (Aug. 1983), 572–581.
11. Scott, D.S. Outline of a mathematical theory of computation. Proc.4th Annual Princeton Conf. Inf. Sci. Syst., Department of Electrical Engineering, Princeton University, Princeton, NJ, 1970 and Tech. Rpt. PRG-2, Programming Research Group, Oxford University, 1970.

CR Categories and Subject Descriptors: D.1.4 [**Programming Techniques**]: Sequential Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; D.2.4 [**Software Engineering**]: Verification—*correctness proofs*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs—*logics of programs*

General Terms: Design, Theory, Verification
Additional Key Words and Phrases: predicative programming

Received 11/82; accepted 8/83

Author's Present Address: Eric C. R. Hehner, Computer Systems Research Group, Sandford Fleming Building, University of Toronto, Toronto, Ontario M5S 1A4, Canada